Data Structures and Algorithms (DSA)

Course 13
Algorithms

Iulian Năstac

Recapitulation
The main application of the heap concept: A sorting technique (**heapsort**)

- The **heapsort** algorithm was invented by J. W. J. Williams in 1964. In the same year, R. W. Floyd published an improved version that could sort an array in-place, continuing his earlier research into the treesort algorithm.

```
heapsort (T[1..n])
{
    make_heap(T);
    for( i = n; i ≥ 2; i - -)
        {
         T[1] ↔ T[i];
         sift_down (T[1…i-1], 1)
        }
}
```

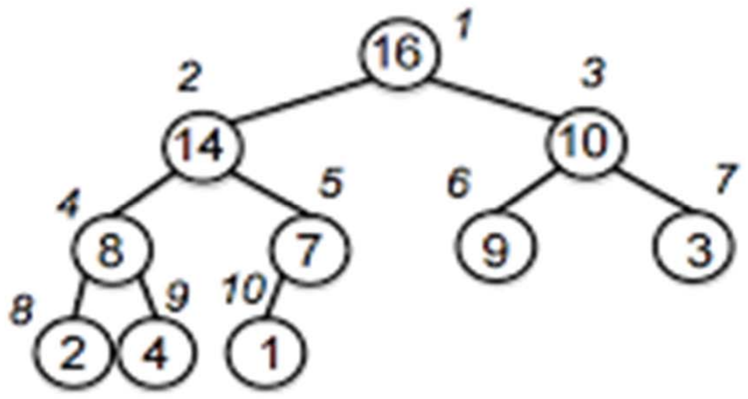
Example

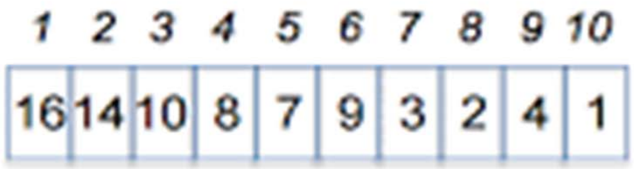
First task is to obtain a heap tree from an unsorted vector

Start from the following vector (which is not a heap):

4	**1**	**3**	**2**	**16**	**9**	**10**	**14**	**8**	**7**
T[1]	**T[2]**	**T[3]**	**T[4]**	**T[5]**	**T[6]**	**T[7]**	**T[8]**	**T[9]**	**T[10]**

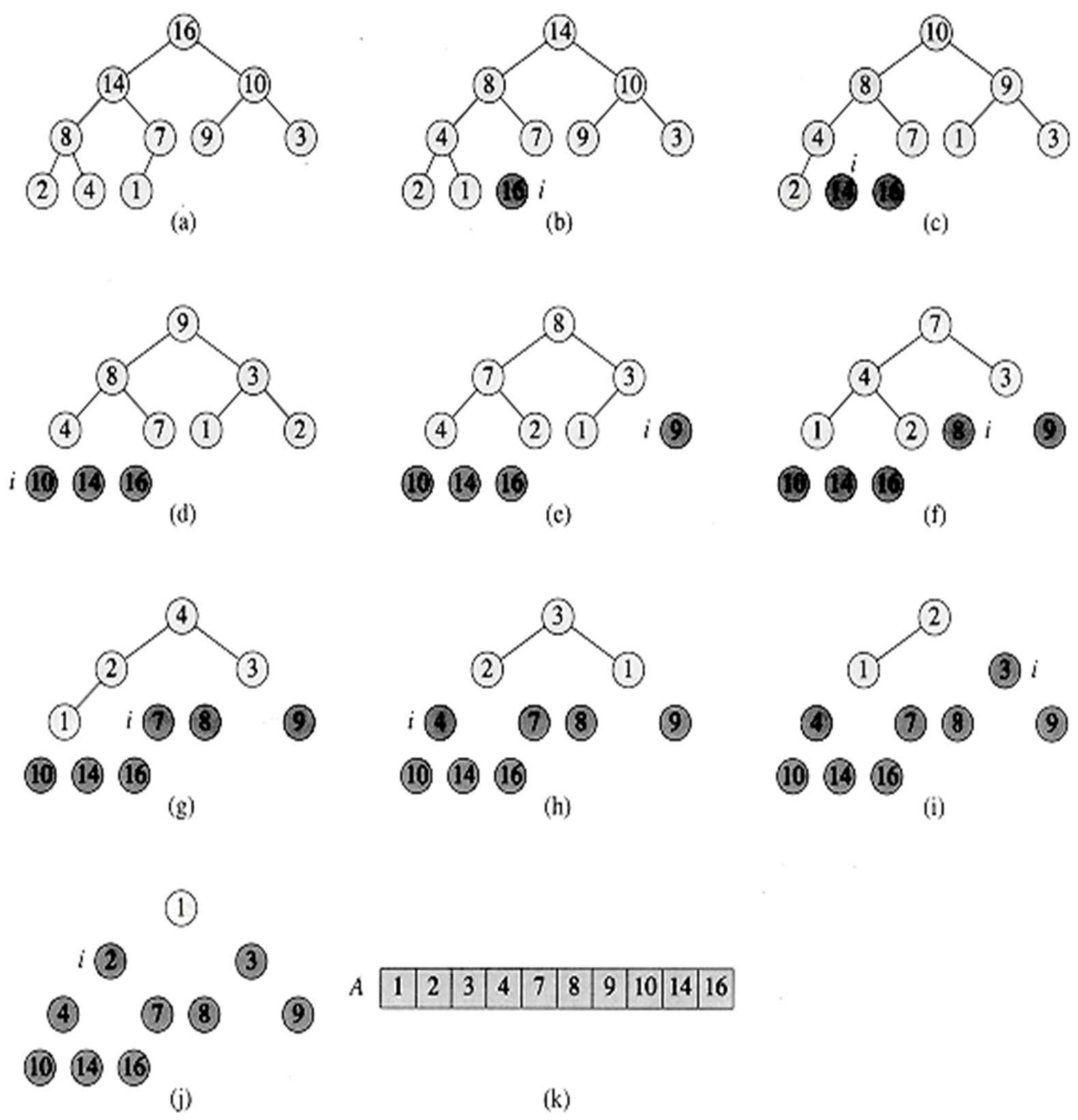
After **make_heap** we obtain:





1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

16	**14**	**10**	**8**	**7**	**9**	**3**	**2**	**4**	**1**
T[1]	**T[2]**	**T[3]**	**T[4]**	**T[5]**	**T[6]**	**T[7]**	**T[8]**	**T[9]**	**T[10]**



The operation of HEAPSORT:

(a) The heap data structure just after it has been built by **make_heap**.

(b)-(j) The heap just after each call of **sift_down(T[1…i-1], 1)**. The value of i at that time is shown. Only lightly shaded nodes remain in the heap.

(k) The resulting sorted array A

Other ordering (or sorting) algorithms:

- Insertion
- Selection
- Merge sort
- Quicksort
- etc.

Ordering methods

- Ordering = arranging items of the same kind, class or nature, in an ordered sequence.
- For this purpose we consider that the data are a collection of items of a certain type and each item comprises one or even more values (variables), which are decisive in the ordering that is performed. Such a value is called **key**.

For the C programming language, a sorting algorithm can be achieved by one of the following methods:

1. Arranging data (which are sorted) so that their keys will finally correspond to the desired order.

2. By ordering an array of pointers (to the data that must be sorted) in order to form an ordered set.

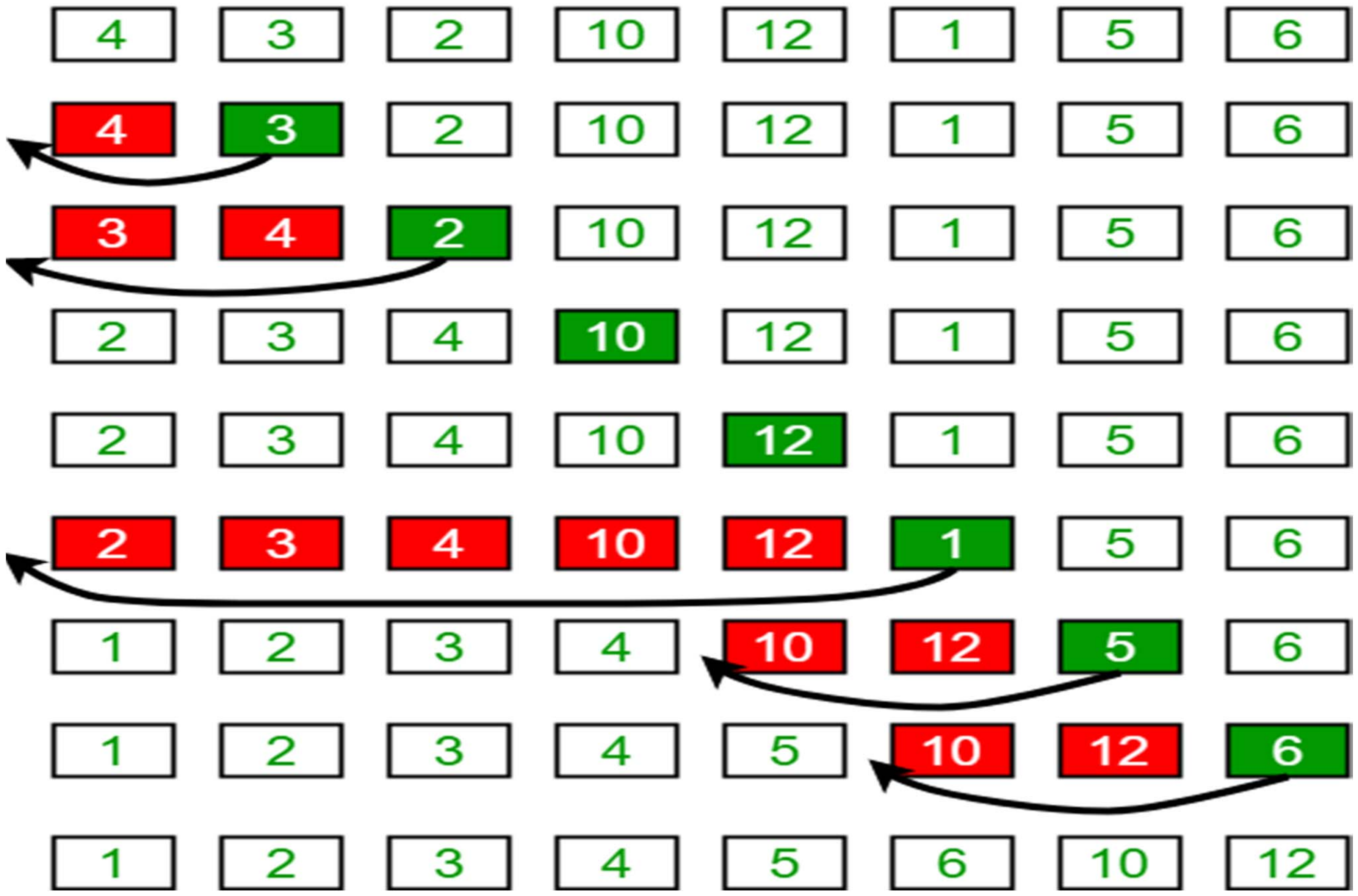
Note:

- In the following we will discuss only about sorting unidimensional arrays (vectors) with numerical data.

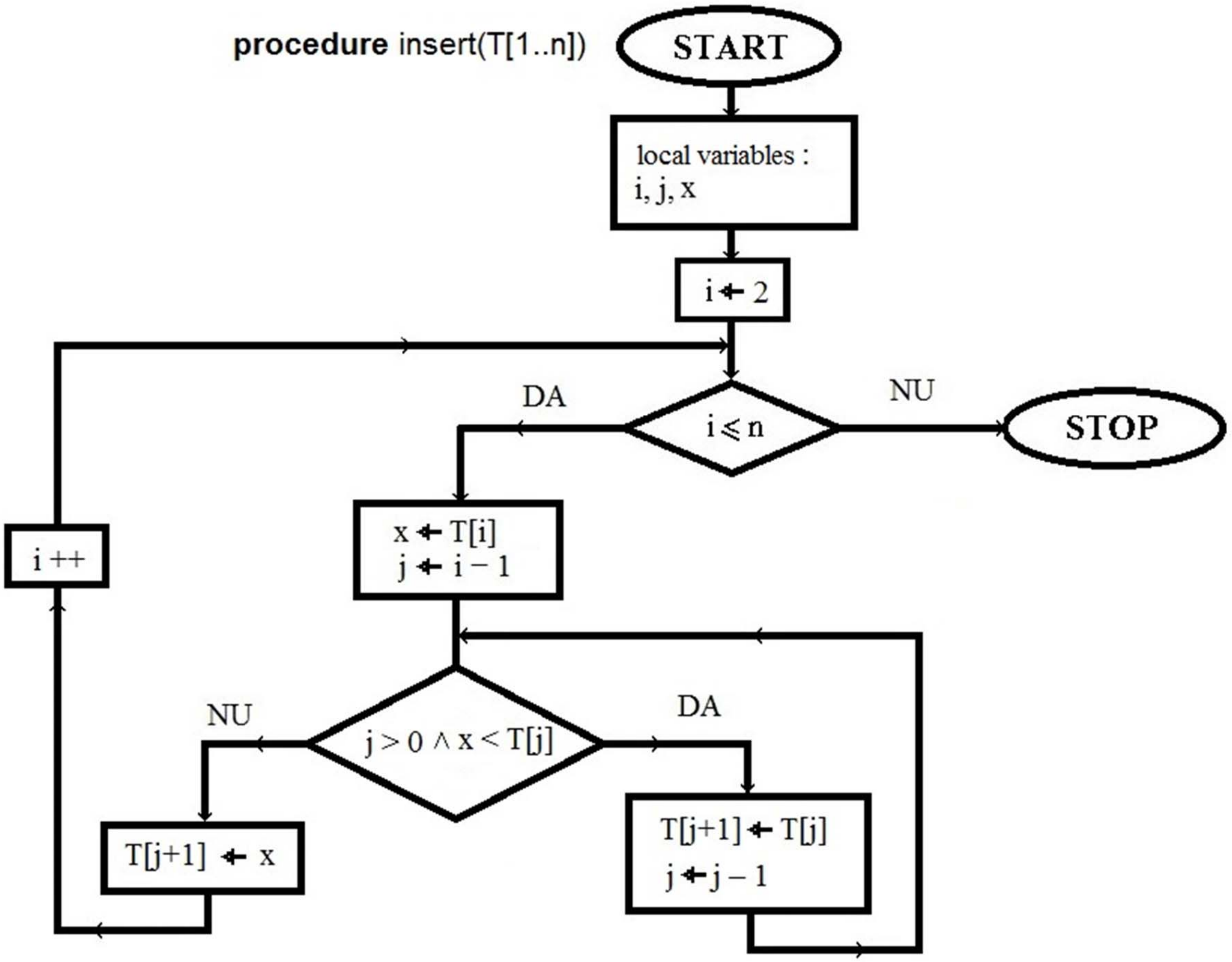
Insertion sorting algorithm
(Recapitulation)

- Insertion sort is a simple sorting algorithm that builds the final sorted array (or list) one item at a time.
- The general idea of sorting by insertion is to consider (at a time) each element of the array and insert it into the substring previously ordered.
- The operation involves a growing sequence, which is moved to the right.

Insertion Sort Execution Example



procedure insert(T[1..n])



A brief description of the algorithm in pseudocode is as follows:

```
procedure insert(T[1..n])
{
          local variables  i, j, x
          for i ← 2 to n do
          {
                    x ← T[i]
                    j ← i -1
                    while (j>0 and x<T[j]) do
                              {
                               T[j+1] ← T[j]
                                j ← j – 1
                              }
                    T[j+1] ← x
          }
}
```

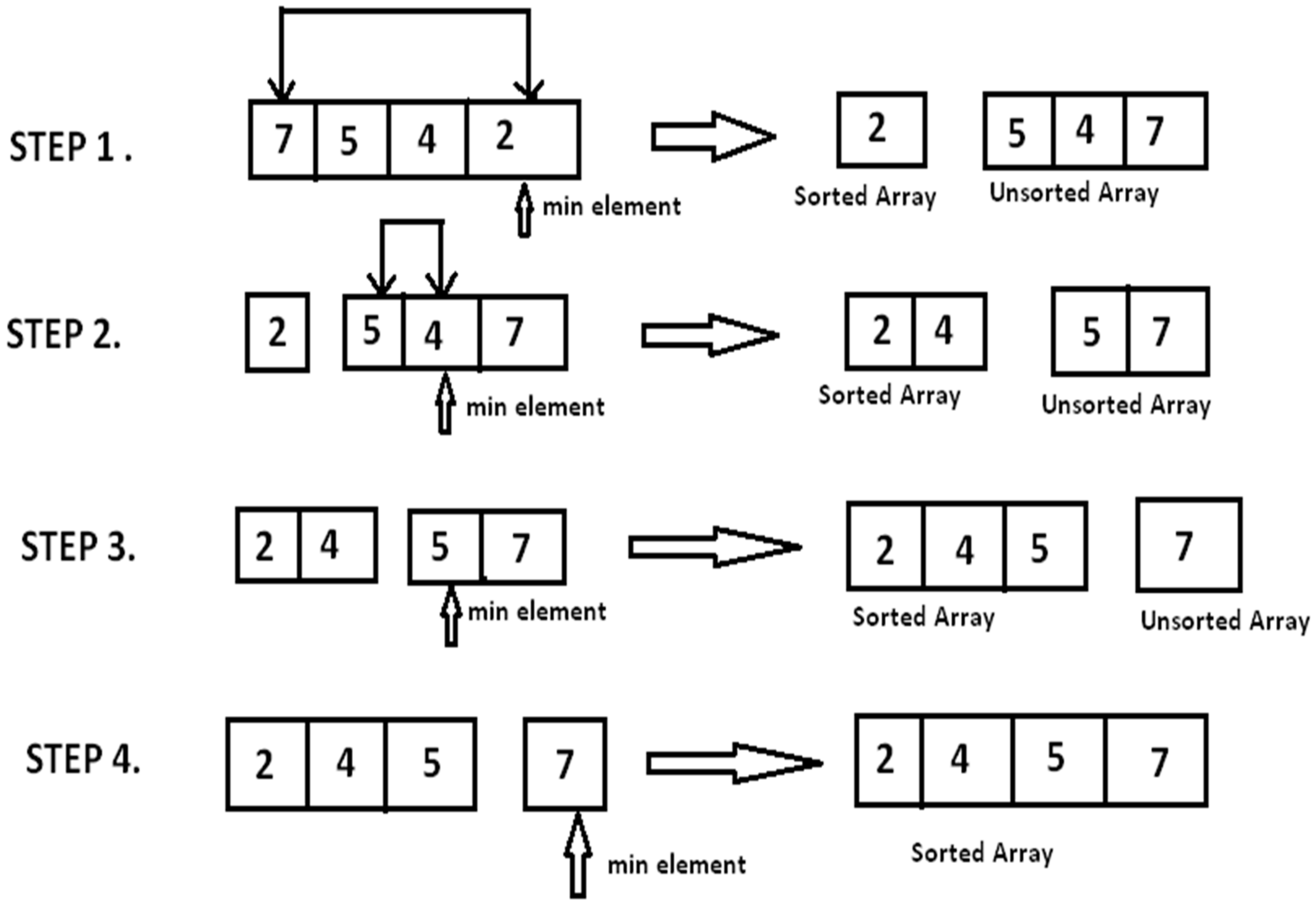
<u>Note</u>: In implementing the above algorithm in C code, we have to keep in mind that a vector is starting with an zero index.

The selection sorting algorithm
(Recapitulation)

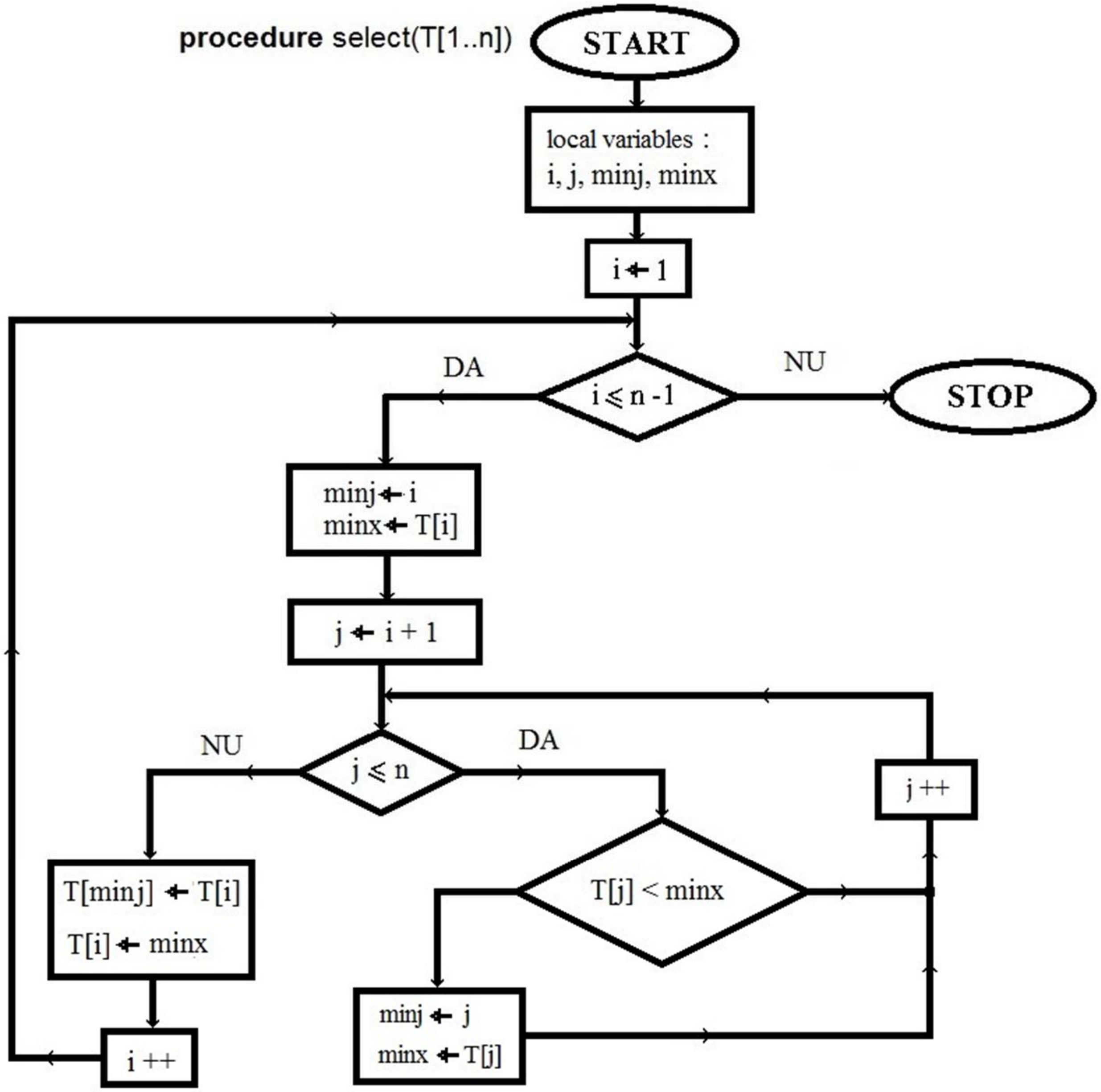
- Selection sort is a sorting algorithm, specifically an in-place comparison sort.
- Selection sort is noted for its simplicity, and it has performance advantages over more complicated algorithms in certain situations, particularly where auxiliary memory is limited.

How it works

- The algorithm divides the input list into two parts: the sublist of items already sorted, which is built up from left to right at the front (left) of the list, and the sublist of items remaining to be sorted that occupy the rest of the list.
- Initially, the sorted sublist is empty and the unsorted sublist is the entire input list.
- The algorithm proceeds by finding the smallest (or largest, depending on sorting order) element in the unsorted sublist, exchanging it with the leftmost unsorted element (putting it in sorted order), and moving the sublist boundaries one element to the right.


STEP 1 .
7
5
4
2
min element
2
Sorted Array
5
4
7
Unsorted Array
STEP 2.
2
5
4
7
min element
2
4
Sorted Array
5
7
Unsorted Array
STEP 3.
2
4
5
7
min element
2
4
5
Sorted Array
7
Unsorted Array
STEP 4.
2
4
5
7
min element
2
4
5
7
Sorted Array

procedure select(T[1..n])



A brief description of the algorithm in pseudocode is as follows:

```
procedure select(T[1..n])
{
  local variables  i, j, min_j, min_x,
  for i ← 1 to n-1 do
          { min_j ← i
            min_x ← T[i]
            for j ← i+1 to n do
                    if T[j] < min_x then
                            { min_j ← j
                              min_x ← T[j]
                            }
            T[min_j] ← T[i]
            T[i] ← min_x
          }
}
```

Note: In implementing the above algorithm in C code, we have to keep in mind that a vector is starting with an zero index.

Bubblesort algorithm (Recapitulation)

- Bubble sort is a simple sorting algorithm that repeatedly steps through the list to be sorted, compares each pair of adjacent items and swaps them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted.
- The algorithm, which is a comparison sort, is named for the way smaller elements "bubble" to the top of the list.
- Although the algorithm is simple, it is too slow and impractical for most problems even when compared to insertion sort.

Shellsort algorithm
(Recapitulation)

- Shellsort is an in-place comparison sort. It can be seen as either a generalization of sorting by exchange (bubble sort) or sorting by insertion (insertion sort).
- The method starts by sorting pairs of elements far apart from each other, then progressively reducing the **gap** between elements to be compared. Starting with far apart elements can move some out-of-place elements into position faster than a simple nearest neighbor exchange.
- Donald Shell published the first version of this sort in 1959.

- The running time of Shellsort is heavily dependent on the **gap** sequence it uses.
- For many practical variants, determining their time complexity remains an open problem.

***Shell sort method can be defined as in the following* :**

1) Start with a **gap = n/2**, where n is the number of the elements that will be sorted.
2) Make a crossing of the vector of items that are sorted.
3) The gap it halves **gap** = **gap/2** .
4) If **gap > 0**, then jump to step 2, otherwise the algorithm stops.

Note *Each crossing through the elements involves the following substeps*:

1) **i = gap**.
2**) j = i – gap + 1** .
3) If **j > 0** and the elements from the positions: **j** and **j+gap** are not ordered, then we will interchange their values. Otherwise jump to substep 6.
4) **j = j – gap**.
5) Jump to substep 3.
6) **i = i + 1**.
7) If **i > n**, the crossing is stopped. Otherwise jump to substep 2.

Notes:

- In implementing the above algorithm in C code, we have to remember that a vector is starting with the zero index.
- Therefore, the initialization from the substep 2) becomes: **j = i – gap** .

A general perspective

- In computer science an algorithm is a self-contained step-by-step set of operations to be performed.
- Algorithms perform calculation, data processing, and/or automated reasoning tasks.
- Practically, there is an endless variety of algorithms (but it is quite useful to classify them - see related books, written by Donald Knuth).

An algorithm can be easily expressed by using a flow chart

- An algorithm is an effective method that can be expressed within a finite amount of space and time and in a well-defined formal language for calculating a function.

- Starting from an initial state and initial input (perhaps empty), the instructions describe a computation that, when executed, proceeds through a finite number of well-defined successive states, eventually producing "output" and terminating at a final ending state.

- The transition from one state to the next is not necessarily deterministic; some algorithms, known as randomized algorithms, incorporate random input.

Other classical algorithms

- **Divide and conquer algorithms**
  - Divide and conquer (D&C) is an algorithm design paradigm based on multi-branched recursion. A divide and conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same (or related) type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.

- **Greedy algorithm**
  - A greedy algorithm is an algorithm that follows the problem solving heuristic of making the locally optimal choice at each stage with the hope of finding a global optimum. In many problems, a greedy strategy does not in general produce an optimal solution, but nonetheless a greedy heuristic may yield locally optimal solutions that approximate a global optimal solution in a reasonable time.

Divide and conquer algorithm

- This technique is the basis of efficient algorithms for all kinds of problems, such as sorting (e.g. quicksort, merge sort, etc.), multiplying large numbers, syntactic analysis (e.g., top-down parsers), and computing the discrete Fourier transform (DFT).
- The name "divide and conquer" is sometimes applied also to algorithms that reduce each problem to only one sub-problem, such as the binary search algorithm for finding a record in a sorted list (or its analog in numerical computing, the bisection algorithm for root finding).

Greedy algorithm

- A greedy strategy for the traveling salesman problem (which is of a high computational complexity) is the following heuristic: "At each stage visit an unvisited city nearest to the current city". This heuristic need not find a best solution, but terminates in a reasonable number of steps; finding an optimal solution typically requires unreasonably many steps. In mathematical optimization, greedy algorithms solve combinatorial problems (of NP complexity).

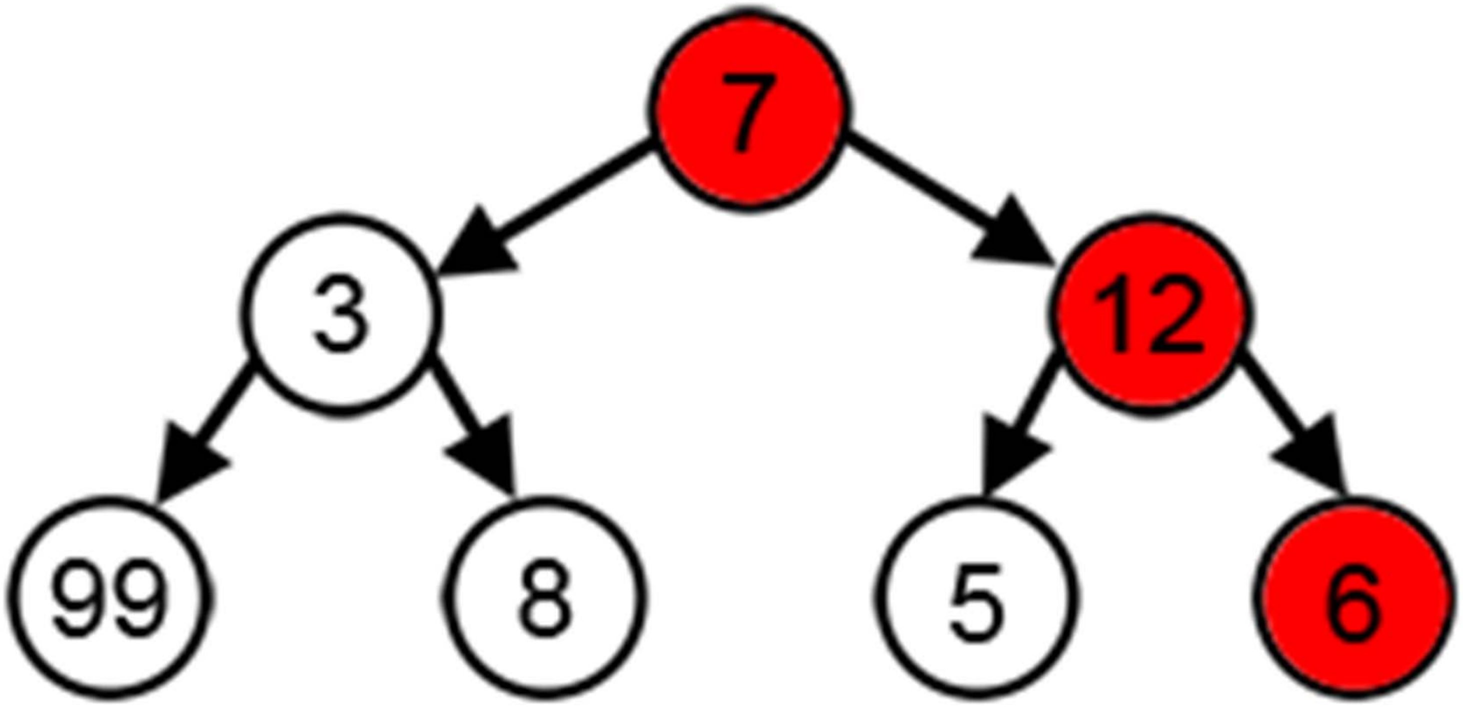
Notes:

- We can make whatever choice seems best at the moment and then solve the subproblems that arise later.
- The choice made by a greedy algorithm may depend on choices made so far but not on future choices or all the solutions to the subproblem.
- It iteratively makes one greedy choice after another, reducing each given problem into a smaller one. In other words, a greedy algorithm never reconsiders its choices.
- This is the main difference from dynamic programming, which is exhaustive and is guaranteed to find the solution.
- After every stage, dynamic programming makes decisions based on all the decisions made in the previous stage, and may reconsider the previous stage's algorithmic path to solution.

Warning!

- Greedy algorithms mostly (but not always) fail to find the globally optimal solution, because they usually do not operate exhaustively on all the data.
- They can make commitments to certain choices too early which prevent them from finding the best overall solution later.
- For example, all known greedy coloring algorithms for the graph coloring problem and all other NP-complete problems do not consistently find optimum solutions.
- Nevertheless, they are useful because they are quick to think up and often give good approximations to the optimum.

Greedy Algorithm



With a goal of reaching the largest-sum, at each step, the greedy algorithm will choose what appears to be the optimal immediate choice, so it will choose 12 instead of 3 at the second step, and will not reach the best solution, which contains 99.

Perspective

- Greedy algorithms are often used in ad hoc mobile networking to efficiently route packets with the fewest number of hops and the shortest delay possible.
- They are also used in machine learning, business intelligence (BI), and artificial intelligence (AI).

Backtracking

- **Backtracking is a general algorithm for finding all solutions of a problem of calculation algorithm that is based on building incremental candidate solutions, each candidate partially abandoned as soon as it becomes clear that he has no chance to be a valid solution.**

- See the last laboratory…

http://www.euroqual.pub.ro/wp-content/uploads/sda_lab_06_backtracking.pdf

Backtracking

- Backtracking depends on:

  - user-given "black box procedures" that define the problem to be solved,

  - the nature of the partial candidates,

  - how they are extended into complete candidates.

It is therefore a metaheuristic rather than a specific algorithm – although, unlike many other meta-heuristics, it is guaranteed to find all solutions to a finite problem in a bounded amount of time.

Analysis of algorithms efficiency

- The analysis of algorithms is the determination of the amount of resources, which are necessary to execute them.

Resources mean:

- ***The memory space*** required for storing the data, which are being processed by the algorithm.
- ***The time*** required for execution of all specified processes of the algorithm.

The study of an algorithm includes several aspects:

- **Designing**
- **Implementation**
- **Validation**
- **Testing:**
  - **Debugging**
  - **Traceability** – refers to the ability to link all product requirements back to the designing plan (relating to code, and test cases).
- **Efficiency evaluation**

Testing Problem

- It can be particularly difficult for large software, with many lines of codes.
- Usually, the cases used to test can not cover all possible situations.
- It requires considerable time and significant computation resources.

“ Program testing can at best show the presence of errors but never their absence…

Nothing is as expensive as making mistakes.”

Edsger Wybe Dijkstra

The efficiency of algorithms

- Ideally, for a given problem, is to find several algorithms and then choose the best one among them …

- … i.e. to find the most efficient algorithm at the current time (using available resources).

- The algorithmic efficiency is very related to the amount of resources used by that algorithm.

An algorithm can be analyzed in two ways:

- **Posteriori** (empirical) - analyze the behavior after the implementation of an algorithm, by running on different cases.

- **A priori** (theoretically) - before implementing and implies quantitative determination of resources (time, memory, etc.)

Notes:

- **The big downside of the posteriori analysis**: practically, an algorithm cannot be tested for any possible extreme cases.
- **A priori analysis**:
  - does not depend on computer or computing language
  - saves time (which could be spent on programming, especially when running an inefficient algorithm)
  - it allows to study the effectiveness for the cases of any size

Improving the computing speed

- Changing an old computer (with a new one, more powerful) may lead to solving a problem 100 times faster (for the same inputs) $\rightarrow$ but this increasing is only a linear one

- only an efficient algorithm can dramatically improve the solving of a problem with huge inputs.

In the following we will be focused only on the time parameter

- The major objective of efficiency is considered the **execution time**.
- Theoretically, other necessary resources (such as the involved memory) may be similarly estimated.

Definition:

- We say that an algorithm requires a time of the order of ***t*** if there is a positive constant ***c***, and an implementation (program code) of the algorithm, capable of solving the problem in each case not more than $c \cdot t(n)$ seconds, where n is the size of the case under consideration.

Note:

The **c** constant actually depends of the CPU's clock frequency.

Asymptotic notation

- We note with:

R_+ - the complete set of all real positive numbers

N - the complete set of all natural numbers

Considering ***$f : N \to R_+$*** as an arbitrary function, we can define the ***O*** notation (or **Big-O** notation):

$$O(f) = \{t : N \to R_+ \mid (\exists c \in R_+^*) \; and \; (\exists n_0 \in N^*) \;\; so \; for \; (\forall n \geq n_0) \;\; we \, obtain \;\; t(n) \leq c \cdot f(n)\}$$

Notes:

- The **Big-O** notation is the formal method of expressing the upper bound of an algorithm's running time. It's a measure of the longest amount of time it could possibly take for the algorithm to complete.

- More formally, for non-negative functions, *f(n)* and *g(n)*, if there exists an integer and a constant $c > 0$ such that for all integers , $f(n) \le c \cdot g(n)$, then *f(n)* is Big O of *g(n)*. This is denoted as "$f(n) = O(g(n))$". If graphed, *g(n)* serves as an upper bound to the curve you are analyzing, *f(n)*.

The principle of invariance

Two different implementations of the same algorithm do not differ (in theirs efficiencies) by more than a multiplicative constant.

Example:

With two different implementations, requiring ***t_1(n)*** seconds, and ***t_2(n)*** seconds respectively, to solve a problem of size n, it results that:

$$\exists c \in R_{+}^{*} \quad so \quad that \quad t_1(n) \leq c \cdot t_2(n) \quad \forall n \geq n_0$$

Note:

$$\boldsymbol{t \in O(t)}$$

Usually we are looking for the simplest function ***f*** so that: $\boldsymbol{t \in O(f)}$

It is obvious that:

$\boldsymbol{n \in O(n)}$, $\boldsymbol{n^2 \in O(n^2)}$, $\boldsymbol{n^2 \in O(n^3)}$, $\boldsymbol{n^3 \notin O(n^2)}$

We can say that:

$$\forall\ f : N \rightarrow R_+^* \ \ if \ \ f \in O(n) \ \Rightarrow \ f^2 \in O(n^2)$$

$$\forall\ f : N \rightarrow R_+^* \ \ if \ \ f \in O(n) \ \Rightarrow \ 2^f \in O(2^n)$$

We can define relations for partial order:

- $f \leq g$ if $O(f) \subseteq O(g)$

- $f < g$ if $O(f) \subset O(g)$

Properties:

P1: Transitivity

If $f \in O(g)$ and $g \in O(h)$ then $f \in O(h)$

P2: If $f \in O(g)$ then $O(f) \subseteq O(g)$

P3: $\forall \; f \;\; and \;\; g : N \rightarrow R_{+}^{*} \; we \; obtain:$

a) $\boldsymbol{O(f) = O(g) \;\Leftrightarrow\; f \in O(g)}$ and $\boldsymbol{g \in O(f)}$

b) $\boldsymbol{O(f) \subset O(g) \;\Leftrightarrow\; f \in O(g)}$ and $\boldsymbol{g \notin O(f)}$

P4:

$$O(f + g) = O(\max(f, g)) \;\; for \; \forall \, f \; and \;\; g : N \rightarrow R_{+}^{*}$$

where the sum and the product, also, are punctualy considered.

We can define the equivalence relation as:

$$\boldsymbol{f \equiv g} \quad \text{if} \quad \boldsymbol{O(f) = O(g)}$$

Note: In the notation ***O(f)*** we can replace ***f*** with other equivalent function.

Example:

lg(n) ≡ ln(n) ≡ log$_2$(n) or ***O(lg(n))=O(ln(n))=O(log$_2$n)***

This is obvious, since it is always possible to find a constant ***c*** so that: ***lg(n) ≤ c·ln(n)***

Since:

$$\log_a x = \frac{\log_b x}{\log_b a}$$

it result that:

$$c \geq \frac{\lg(n)}{\ln(n)} = \frac{\frac{\ln(n)}{\ln(10)}}{\ln(n)} = \frac{1}{\ln(10)}$$

If we denote by O(1) the higher order of the functions bounded by a constant, then we can obtain the hierarchy:

$$O(1) \subset O(\log n) \subset O(n) \subset O(n \cdot \log n) \subset O(n^2) \subset O(n^3) \subset ... \subset O(2^n)$$

This hierarchy corresponds to a performance criterion.

For a given problem, we always want to obtain a corresponding algorithm with its order as much as possible to the left side in the specified hierarchy.

Example:

$$n^3 + 3 \cdot n^2 + n + 8 \in O(n^3 + (3 \cdot n^2 + n + 8)) = O(max(n^3 , (3 \cdot n^2 + n + 8)))$$
$$= O(n^3)$$

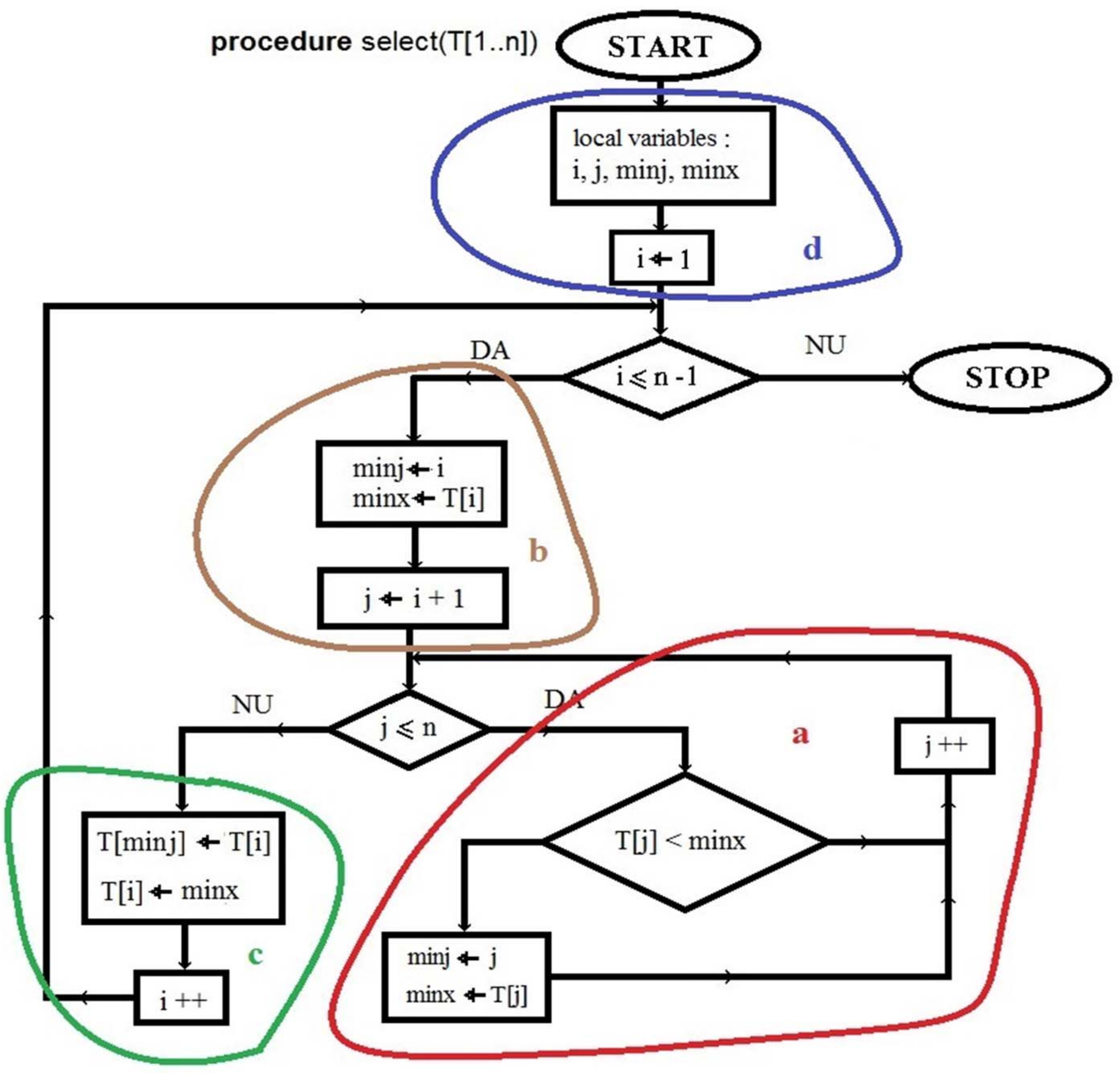
Note:

$max(n^3 , (3 \cdot n^2 + n + 8))$ will be ***n^3*** for a number, which is sufficient great (since the increase of the chart for n^3 is faster than any second-degree polynomial even if it is started from lower values).

Algorithms analysis techniques

- There isn't a general formula for analyzing the efficiency of algorithms.
- This is a matter of judgment, intuition and experience.
- The flowchart may be useful in order to find the Big-O of an algorithm.

Selection sort



- The time for a single execution of the inner loop can be upper bounded by an ***a*** constant.

- For a given ***i***, the inner loop needs a time frame of not more than

$$\boldsymbol{b + a \cdot (n - i)} \text{ units}$$

where ***b*** is a constant that denotes the loop initialization.

- A single execution of the external loop needs no more than

$$\boldsymbol{c + b + a \cdot (n - i)} \text{ units}$$

where ***c*** is other constant.

Finally, the total time for the algorithm is no more than:

$$T_{\max} = d + \sum_{i=1}^{n-1} (c + b + a(n - i)) \quad \text{units}$$

$$T_{\max} = a \cdot n \cdot \sum_{i=1}^{n-1} 1 - a \cdot \sum_{i=1}^{n-1} i + (c+b) \cdot (n-1) + d$$

but:

$$\sum_{i=1}^{n-1} i = \frac{(n-1+1) \cdot (n-1)}{2} = \frac{n^2}{2} - \frac{n}{2} \quad \text{and} \quad \sum_{i=1}^{n-1} 1 = n-1$$

It result:

$$T_{\max} = \tfrac{a}{2} \cdot n^2 - \tfrac{a}{2} \cdot n + (b+c-a) \cdot (n-1) + d - c - b$$

Therefore the insertion sorting algorithm requires a runtime of ***O(n²)***